Nim's Garbage Collector 0.16.1

Andreas Rumpf

April 27, 2017

"The road to hell is paved with good intentions."

1 Introduction

This document describes how the GC works and how to tune it for (soft) realtime systems.

The basic algorithm is *Deferred Reference Counting* with cycle detection. References on the stack are not counted for better performance (and easier C code generation). Cycle detection is currently done by a simple mark&sweep GC that has to scan the full (thread local heap). `-gc:v2` replaces this with an incremental mark and sweep. That it is not production ready yet, however.

The GC is only triggered in a memory allocation operation. It is not triggered by some timer and does not run in a background thread.

To force a full collection call `GC_fullCollect`. Note that it is generally better to let the GC do its work and not enforce a full collection.

2 Cycle collector

The cycle collector can be en-/disabled independently from the other parts of the GC with `GC_enableMarkAndSweep` and `GC_disableMarkAndSweep`.

3 Realtime support

To enable realtime support, the symbol useRealtimeGC needs to be defined via `-define:useRealtimeGC` (you can put this into your config file as well). With this switch the GC supports the following operations:

```
proc GC_setMaxPause*(maxPauseInUs: int)
proc GC_step*(us: int, strongAdvice = false, stackSize = -1)
```

The unit of the parameters `maxPauseInUs` and `us` is microseconds.

These two procs are the two modus operandi of the realtime GC:

(1) GC_SetMaxPause Mode

You can call `GC_SetMaxPause` at program startup and then each triggered GC run tries to not take longer than `maxPause` time. However, it is possible (and common) that the work is nevertheless not evenly distributed as each call to `new` can trigger the GC and thus take `maxPause` time.

(2) GC_step Mode

This allows the GC to perform some work for up to `us` time. This is useful to call in a main loop to ensure the GC can do its work. To bind all GC activity to a `GC_step` call, deactivate the GC with `GC_disable` at program startup. If `strongAdvice` is set to `true`, GC will be forced to perform collection cycle. Otherwise, GC may decide not to do anything, if there is not much garbage to collect. You may also specify the current stack size via `stackSize` parameter. It can improve performance, when you know that there are no unique Nim references below certain point on the stack. Make sure the size you specify is greater than the potential worst case size.

These procs provide a "best effort" realtime guarantee; in particular the cycle collector is not aware of deadlines yet. Deactivate it to get more predictable realtime behaviour. Tests show that a 2ms max pause time will be met in almost all cases on modern CPUs (with the cycle collector disabled).

3.1 Time measurement

The GC's way of measuring time uses (see `lib/system/timers.nim` for the implementation):

1. `QueryPerformanceCounter` and `QueryPerformanceFrequency` on Windows.
2. `mach_absolute_time` on Mac OS X.
3. `gettimeofday` on Posix systems.

As such it supports a resolution of nanoseconds internally; however the API uses microseconds for convenience.

Define the symbol `reportMissedDeadlines` to make the GC output whenever it missed a deadline. The reporting will be enhanced and supported by the API in later versions of the collector.

3.2 Tweaking the GC

The collector checks whether there is still time left for its work after every `workPackage`'th iteration. This is currently set to 100 which means that up to 100 objects are traversed and freed before it checks again. Thus `workPackage` affects the timing granularity and may need to be tweaked in highly specialized environments or for older hardware.

3.3 Keeping track of memory

If you need to pass around memory allocated by Nim to C, you can use the procs `GC_ref` and `GC_unref` to mark objects as referenced to avoid them being freed by the GC. Other useful procs from system you can use to keep track of memory are:

- getTotalMem(): returns the amount of total memory managed by the GC.
- getOccupiedMem(): bytes reserved by the GC and used by objects.
- getFreeMem(): bytes reserved by the GC and not in use.

In addition to `GC_ref` and `GC_unref` you can avoid the GC by manually allocating memory with procs like `alloc`, `allocShared`, or `allocCStringArray`. The GC won't try to free them, you need to call their respective *dealloc* pairs when you are done with them or they will leak.

4 Heap dump

The heap dump feature is still in its infancy, but it already proved useful for us, so it might be useful for you. To get a heap dump, compile with `-d:nimTypeNames` and call `dumpNumberOfInstances` at a strategic place in your program. This produces a list of used types in your program and for every type the total amount of object instances for this type as well as the total amount of bytes these instances take up. This list is currently unsorted! You need to use external shell script hacking to sort it.

The numbers count the number of objects in all GC heaps, they refer to all running threads, not only to the current thread. (The current thread would be the thread that calls `dumpNumberOfInstances`.) This might change in later versions.